HETEROGENEOUS DATA TRANSLATION
THROUGH XML CONVERSION

PAOLO PAPOTTI and RICCARDO TORLONE
Dip. di Informatica e Automazione
Università Roma Tre
00146, Roma, Italy
\{papotti,torlone\}@dia.uniroma3.it

Received November 5, 2004
Revised May 3, 2005

In this paper, we illustrate an approach to the translation of Web data between heterogeneous formats. This work fits into a larger project whose aim is the development of a tool for the management of data described according to a large variety of formats. As an example, the automatic translation of schemes and instances from one model to another is considered [3]. Data translations operate over XML representations of schemes and instances, and they rely on a uniform description of models that we call metamodel. The metamodel shows structural diversities and dictates the needed transformations. Complex translations are derived automatically by combining a number of predefined basic procedures. These procedures perform XML transformations and are implemented by means of XML query languages. Practical examples are provided to show the effectiveness of the approach.

Keywords: Data models, metamodel, data translation, heterogeneous databases, XML.

1 Introduction

Very often today, data cooperation and interchange between different organizations is made difficult by the fact that little or no advance standardization exists and data is stored under different formats in distinct heterogeneous Web data sources [1]. Therefore, the need arises for an integrated management of heterogeneous descriptions of data that allow for easy and flexible translations from one format to another [6]. This problem is related to, but different from, the problems of data integration [4] and schema matching [30]. Recently, various aspects of the data translation problem have been studied in the context of the relational model [15, 16] and in even more general settings [24, 27, 28]. However, it is widely recognized that there is still a compelling need for a general solution able to cope, in a uniform way, the large diversity of the various formats available [5].

In this framework, the final goal of our research project is the development of a tool for: (i) the management of data available on the Web described according to a large variety of formats and models, and (ii) the automatic translation of schemes and instances from one model to another. The tool can be seen as an implementation of the "ModelGen" operator proposed by Bernstein in the context of Model Management Systems [5].

The set of models managed by the tool we have in mind should include the majority
of the formats used to represent data in Web-based applications: semi-structured models, schema languages for XML, specific formats for, e.g., scientific data, as well as conceptual data models. Actually, the set of models is not fixed a priori in this environment: a facility allows expert users to define a new model $M$ at run-time and translations for $M$ are derived by the system with limited or none user intervention.

As a first result of our project, we have proposed a preliminary tool for the management and the automatic translation of schemes between formats and models used to represent data in Web-based applications [35, 36]. The approach relies on a revised notion of metamodel, which we have introduced in an earlier work in the context of the management of multiple models in a database design tool [3]. A metamodel is a formalism that allows the uniform representation of models and the identification of differences between primitives used in the various models. It is expressed in XML and embeds, on the one hand, the main primitives adopted by different schema languages for XML [21] and, on the other hand, the basic constructs of traditional database conceptual and logical models [19]. Translations are automatically derived by combining a set of predefined procedures that operate over individual primitives and implement standard transformations.

In this paper, we present an approach that, building on the translation derived at scheme-level, has the goal of generating a corresponding translation at instance-level. Source data is first serialized and represented in XML. Then, XML data is restructured to conform to the constructs allowed in the target model. Finally, it is deserialized into the specific syntax of the target. The restructuring phase is the more involved step and is performed by combining a number of predefined basic functions expressed in XQuery [17]. The effectiveness of the approach is demonstrated by a number of practical experiments on common cases that often arise in practice.

The system makes use of XML for several reasons. First, XML allows for a natural and flexible description of information at different levels of abstraction. Moreover, XML is today a widely accepted standard for data exchange. For this reason, almost all data management tools provide today an import/export facility able to convert data from the internal representation to XML and vice versa. It follows that the serialization/deserialization steps of the translation procedure described above can be often demanded to external systems.

The rest of the paper is organized as follow. In Section 2 we provide a general overview of our approach to model management. In Section 3 we present a novel technique for data translation between different models and in Section 4 we illustrate a complete example of translation. In Section 5 we compare our approach with relevant literature and finally, in Section 6, we discuss some open issues and sketch future directions of research.

2 An overview of the approach

2.1 Basics

Let us first clarify our terminology. In our framework, we identify four levels of abstractions. At the bottom level we have actual data (or instances) organized according to a variety of (semi) structured formats (relational tables, XML documents, HTML files, scientific data, and so on). At the second level we have schemes, which describe the structure of the instances (a relational schema, a DTD, an XML Schema or one of its dialects [21], etc.). Then, we have different formalisms for the description of schemes that we call models hereinafter
(e.g., the relational model, the XML Schema model or even a conceptual model like the ER model). Finally, we use the term *metamodel* to mean a general formalism for the definition of the various models. Specifically, our *metamodel* is made of a set of *metaprimitives*. Each metaprimitive captures a class of constructs of different data models that share common characteristics or, more precisely, that implement, possibly with different names, the same basic abstraction principle [35]. Examples of metaprimitives are: class, attribute, base domain, relationship, set, (ordered) list, generalization, disjoint union, key, foreign key, and so on. In this framework, a model is defined as a set of *primitives*, each of which is classified according to a metaprimitive of the metamodel. For instance the relational model provides the *table* primitive that is an instance of the metaprimitive *relation* over basic domains.

A *translation* is defined as follows: given two models $M_s$ (the *source model*) and $M_t$ (the *target model*) represented by the metamodel, a set of data $D_s$ (the *data source*) of a scheme $S_s$ (the *source scheme*) for $M_s$, a *translation of* $D_s$ and $S_s$ into $M_t$ is a set of data $D_t$ (the *data target*) of a scheme $S_t$ (the *target scheme*) for $M_t$ representing the same information as $D_s$.

### 2.2 The translation technique

Let $M_s$ be the *source model*, $M_t$ the *target model* and $D_s$ be the *data source* described by a scheme $S_s$. The translation of $D_s$ into $M_t$ relies on an internal concept, called *supermodel*, that is used as a reference in the translation [35]. A supermodel is a special model $M^*$, maintained automatically by the system, that has one primitive for each metaprimitive of the metamodel and is the therefore the most expressive model that can be represented with the metamodel. Since, by construction, for each primitive of $M_s$ ($M_t$) there is a corresponding primitive in $M^*$, it follows that, syntax apart, both $M_s$ and $M_t$ are subsets of $M^*$. Therefore, any scheme that is valid for $M_s$ or $M_t$ is also valid for $M^*$. The schemes of the supermodel are expressed in an XML-based syntax that makes use of the metaprimitives as tags.

The translation technique is composed by a number of steps as follows.

1. A plain XML conversion that preserves the original structure is performed on both $D_s$ and $S_s$. As we have said in the introduction, this task is usually supported by the source system in which $D_s$ is stored. The output is a set $\hat{D}_s$ of XML data and an XML representation of the scheme $\hat{S}_s$ of $\hat{D}_s$ (e.g., an XML Schema). An example of serialization of a relational table, as it can be generated by a commercial system, is reported on the top of Figure 1.

2. The scheme $\hat{S}_s$ is translated into the supermodel. This is actually a rather simple task that just requires a renaming of constructs. The output is a scheme $\hat{S}^*_s$ of the supermodel. The rational under this step is that in this way $\hat{S}^*_s$ can be easily matched against the target model, which is a subset of the supermodel. Step 2 of Figure 1 reports the translation of the XML Schema representing a relational data store into the supermodel.

3. The scheme $\hat{S}^*_s$ is restructured by translating primitives used in the source scheme that are not allowed in the target model. The output of this operation is a scheme $\hat{S}^*_t$ of the

---

We stress the fact that we are interested in the *translation* of a data set into a different representation rather than the integration of heterogeneous data sources or the derivation of a mapping between them.
Fig. 1. A complete translation from the relational model to ODL
supermodel that makes use only of constructs allowed in the target model. Accordingly, the data set \( \hat{D}_i \) is translated into a format \( \hat{D}_t \) that is coherent with \( \hat{S}_t \). Step 3 of Figure 1 reports the translation of scheme and data into an object model within the supermodel. Note that the relational construct has been replaced by a class construct and, for each tuple, an oid (object identifier) has been created, as required in an object-oriented model.

4. The scheme \( \hat{S}_t \) is renamed into a scheme \( \hat{S}_t \) using the syntax of the target model \( M_t \) and finally, both \( \hat{S}_t \) and the data set \( \hat{D}_t \) are deserialized and delivered to the target system. The last step in Figure 1 describes the representation of the target scheme and the target data in ODL, an ODMG standard for object oriented database systems [10].

Step 3 is the crucial point of the translation procedure: it takes as input a data set and its scheme and transforms them in a format suitable for the target model. Since this operation occurs within the supermodel, where each primitive represents a class of constructs from different models, we can apply "generic" transformations that are independent of the particular pair of models at hand. It follows that, as the number of primitives is limited, it is possible to predefine a number of basic transformations that can be composed to build complex translations.

In the next section, we will present in more details our approach to this phase.

3 Translation of models

As we have said in the previous section, the transformation operated within the supermodel is the crucial step of our approach. The other steps are rather easy, can be supported by external systems, and are not always needed when source and/or target are already represented in XML. Therefore, we concentrate our attention on an algorithm for phase 3, where schemes and data, expressed in the supermodel, are restructured according to the target model.

3.1 The translation algorithm

The algorithm takes as input a scheme \( S_s \) of a source model \( M_s \) and the target model \( M_t \), and returns the target scheme \( S_t \) and a transaction \( t \) (that is, a sequence of data manipulation operations) that, applied to any instance \( I_s \) of \( S_s \) generates an instance \( I_t \) of \( S_t \). We recall that in this phase, \( S_s \) is expressed in terms of primitives of the supermodel, that is, in a common language that allows the comparison of constructs of different models.

The algorithm is reported in Figure 2 and proceeds by analyzing the scheme \( S_s \) as follows: for each primitive \( C \) used in \( S_s \), it verifies whether \( C \) is allowed in the target model. If this is not the case, it tries to convert \( C \) into another primitive (or a set thereof) available in the target model. This work is supported by a library \( L \) of predefined basic procedures \( p \) that implement rather standard translations between primitives. Each of these procedures has indeed two components: a schema-level function \( f_S \), which performs translations of primitives, and a function \( f_I \), which operates at instance level by transforming actual data according to the translations operated by \( f_S \). Specifically, these functions must satisfy the following consistency criterium.

**Definition 1 (Consistency of basic procedures)** A basic procedure \( p[f_S, f_I] \) is consistent if, for each scheme \( S \) and for each instance \( I \) of \( S \), \( f_I(I) \) is an instance of \( f_S(S) \).
Algorithm 1

Input: A scheme $S_1$ of a model $M_1$, the target model $M_t$ and a library of procedures $L = \{p_1[f^1_S, f^1_I], \ldots, p_k[f^k_S, f^k_I]\}$

Output: A transaction $t$, a scheme $S_t$ for $M_t$, and the residual $m_t$ for $S_t$

begin
(1) Set a temporary scheme $S$ to the source scheme $S_s$;
(2) Set $t$ to the empty transaction;
(3) while there is a primitive $C$ in $S$ such that $C$ is not allowed in $M_t$ do
(4) if there exists a procedure $p_i[f^i_S, f^i_I]$ in $L$ such that $f^i_S$ translates $C$ to a primitive (or a set thereof) allowed in $M_t$
(5) then /* direct translation */
(6) $S = f^i_S(S)$; /* apply $f^i_S$ to $S$ */
(7) add the residual generated by $f^i_S$ to $m_t$ (if any);
(8) $t = t, f^i_I$; /* append $f^i_I$ to $t$ */
(9) else
(10) if there exists a procedure $p_i[f^i_S, f^i_I]$ in $L$ such that $f^i_S$ translates $C$ to a primitive (or a set thereof) not allowed in $M_t$
(11) then /* try to find an intermediate translation */
(12) $S = f^i_S(S)$; /* apply $f^i_S$ to $S$ */
(13) add the residual generated by $f^i_S$ to $m_t$ (if any);
(14) $t = t, f^i_I$; /* append $f^i_I$ to $t$ */
(15) else
(16) abort the translation and notify the user;
end while
(17) $S_t = S$; /* $S$ becomes the target scheme */
end

Fig. 2. The translation algorithm

The behavior of a basic procedure $p$ is represented by a signature, that is, an abstract description of the primitives on which $p$ operates (the input) and of the primitives generated by $p$ (the output). This signature is used by the algorithm to select the appropriate procedure to apply, without actually executing it. In this way, we make the algorithm independent of the actual implementation of the various procedures. Representatives of such procedures will be presented in more detail in Section 3.3.

When a procedure translates a primitive into another primitive that cannot express information at the same level of detail, we say that it produces a semantic loss. As an example, a procedure may need to transform a generalization between entities into a generic relationship between them, since the target model does not provide a generalization primitive. In this case, the procedure generates extra information, which we call residual, describing this event. The residuals of a translation are collected and stored externally, in a file associated with the target scheme.

There are a number of important aspects to point out about this algorithm.

- In step (4), it may happen that more than one procedure available in $L$ can perform the needed translation. For instance, it is well known that there are several ways to
implement generalizations using other primitives. A possible solution in this case is to request the intervention of the user, in order to make a choice between the various possibilities. Another solution could be to solve ambiguity by introducing a preference order between procedures.

- In step (10), the selected procedure translates a primitive $C$ into a primitive that is actually not allowed in $M_t$. The rationale underlying this operation is that if the algorithm is not able to translate directly into a primitive of $M_t$, it tries to translate $C$ into an intermediate primitive $C'$ that is not allowed in the target model, but that can be translated by another procedure into a primitive $C''$ of the target model. Consider for instance the translation from an object-oriented data model into a DTD representation. Since there is not a direct representation of generalization hierarchies in a DTD, we can first translate them into relationships and then translate relationships into DTD elements and attributes.

- Since the effect of a procedure can be the inverse of another, because of the way in which procedure are selected, the algorithm can enter into infinite loops. In order to prevent this situation, we can introduce an halt condition that occurs when the selected procedure introduces a primitive that were deleted in a previous step. This guarantees the monotonicity of the process. When an halt state occurs, the algorithm backtracks to a step in which a different selection of a procedure can be done.

- Some procedure $p[f_s, f_t]$ of the library $L$ may not require a data translation, that is, $f_t$ is the identity function. Assume, for example, that we need to translate a scheme $S$ with a cardinality constraint of type $(1, 10)$ to a model that allows only cardinalities of the form $(1, 1), (1, n)$ and $(0, n)$. This change operates at schema level but does not affect data. On the other hand, many primitives require data manipulation, like the invention of keys.

As a final comment, we note that the presented algorithm can be improved in several points. In particular, a final optimization step can be introduced on the output transaction by eliminating redundant or useless functions and by finding a better execution order.

### 3.2 Properties of translations

The problem of model translations involves a number of important conceptual questions. A relevant issue is related to the analysis of the quality of a translation. In [3] we have proposed some properties that “good” translations between conceptual models should enjoy. Many of these properties also apply to the framework of this paper.

The basic requirement is the correctness of a translation. It requires that:

1. the output scheme be a valid scheme for the target model, and
2. the output instance be a valid instance for the target scheme.

It is easy to show that correctness is actually guaranteed by the technique presented above. Condition 1 follows from the main algorithm, which progressively substitutes primitives not
allowed in the target, whereas Condition 2 follows by the consistency of basic procedures (Definition 1).

Other important properties are equivalence and minimality. The former requires the
existence of a one-to-one correspondence between the target and the source instances, the
latter expresses the fact that shorter translations do not exist. The investigation of these and
further properties of translations is subject of current work.

3.3 Basic procedures
In this section we illustrate some examples of basic procedures used by the Algorithm reported
in Figure 2. They are used within the supermodel, where the system matches models and
selects metaprimitives to be transformed, as indicated by Algorithm 1. We recall that each
procedure $p$ is composed by two functions: $f_S$, which operates at scheme level, and $f_I$, which
operates at instance level.

1. Nesting of complex elements. This procedure nests a pair of elements according to
a referential integrity constraint between them.
   $f_S$: it nests an element definition $E_1$ into another element definition $E_2$ and deletes the
   corresponding referential integrity constraint.
   $f_I$: it groups and nests instances of $E_1$ into the corresponding instances of $E_2$ and
deletes references between them.

2. Unnesting of complex elements. The procedure flat nested elements and introduces
   integrity constraints between them.
   $f_S$: it takes an element $E_1$ nested into another element $E_2$, moves the definition of $E_1$
at the same level of $E_2$ and introduces an integrity constraint between them.
   $f_I$: it moves instances of $E_1$ outside instances of $E_2$.
   This procedure is discussed in more detail at the end of this section.

3. Key creation. It generates unique identifiers for elements.
   $f_S$: it adds a key constraint $K$ to an element $E$.
   $f_I$: it invents a value for $K$ for each instance of $E$ using either a Skolem functor [20] or
   a unique integer.

4. Removal of namespaces. This procedure removes information on the domain of the
   names used in a scheme.
   $f_S$: it deletes the namespace definition and stores this information in the residual.
   $f_I$: it does not perform any modification on instances.

5. Cardinality range extension. Cardinalities are used at different levels of precision
   in the various models. This procedure changes the actual value of a cardinality to an
   undefined value.
   $f_S$: it enforces a maximum cardinality different from 1 to the undefined value $N$ and
   stores the original value in the residual.
   $f_I$: it does not perform any modification on instances.

6. Cardinality range restriction. Differently from the previous procedure, this proce-
dure implies some involved transformation on the instances.
$f_S$: it modifies values that express a cardinality and stores the original value in the residual.

$f_R$: it groups or splits elements in order to match the new cardinality values.

7. **Transformation of ordered sequences into unordered ones.** The procedure introduces a new attribute that codes the order.

   $f_S$: it introduces a new attribute $N$.

   $f_R$: it assigns to $N$ a positive integer coding the original position of the element in the sequence.

8. **Transformation of unordered sequences into ordered ones.** It enforces an order to an unordered sequence.

   $f_S$: it just substitutes the primitive.

   $f_R$: it enforces an order to members of the sequence according to the order specified in a positional attribute.

9. **Transformation of generalization hierarchies.** The procedure removes generalizations and translates them in other primitives.

   $f_S$: it eliminates a generalization according to a certain policy (e.g., using relationships or grouping elements) and stores the choice in the residual.

   $f_R$: it performs a modification on the instances according to the choice done at scheme level.

10. **Management of built-in types.** Usually, different models have different built-in base types. This procedure modifies schemes and instances according to a substitution of built-in types by using a table of conversions between basic types (strings to integers, integers to decimals and so on).

    $f_S$: it translates the type definitions and stores lost information in the residual.

    $f_R$: it casts values according to the conversion table (e.g., it transforms a number into a string).

11. **Removal of user types.** Some models allow the definition of user types by extending or restricting built-in data types. This procedure removes this feature.

    $f_S$: it deletes a user type definition and stores this information in the residual.

    $f_R$: it casts values defined over user types into values of basic types.

As a concrete example, we now present in more detail the unnesting procedure. Unnesting is a rather common issue in model translation. For instance, it arises when we need to store XML data into a relational database, a problem largely debated in the literature [18]. Here, we just show intuitive algorithms, based on a combination of elementary operations over XML data. All the complex elements must contain a key (this can be guaranteed by the application of the key creation procedure). The first function, in the left hand side of Figure 3, takes as input a scheme $S_1$ and outputs a scheme $S_2$, where nested elements are converted into flat ones. The second is reported in right hand side of the same figure and operates accordingly on data: it takes as input an instance $I_1$ of the scheme $S_1$ and outputs an instance $I_2$ of the scheme $S_2$. 
**Scheme unnesting**

**Input:** A scheme $S_s$ with nested elements and its residual

**Output:** A scheme $S_t$ without nested elements and its residual

```
begin
  set $S$ to the empty scheme;
  for each element $ex$ in $S_s$ do
    if $ex$ is a complex element
      case $ex$ of
        is nested in a complex element $ec$:
          copy $ex$ in $S$ outside $ec$;
        add to $ex$ a foreign key $kr$ for $ec$;
        is not nested:
          copy $ex$ in $S$;
      end case
    else  "ex is atomic"
      case $ex$ of
        is nested in a complex element $ec$:
          copy $ex$ in $S$ inside $ec$;
        is not nested:
          copy $ex$ in $S$;
      end case
  end for
  eliminate intermediate elements;
end
```

**Data unnesting**

**Input:** An instance $I$ with nested data, the scheme $S_s$ of $I$

**Output:** An instance $I_t$ without nested data

```
begin
  set $I$ to the empty instance;
  for each element $ex$ in $I_s$ do
    if $ex$ is a complex element
      case $ex$ of
        is nested in a complex element $ec$:
          copy each occurrence of $ex$ in $I$ outside $ec$;
        add to each occurrence of $ex$ the key of $ec$;
        is not nested and contains an atomic element $ea$:
          copy each occurrence of $ex$ in $I$;
      end case
    else  "ex is atomic"
      case $ex$ of
        is nested in a complex element $ec$:
          copy in $I$ each occurrence of $ea$;
        is not nested:
          eliminate intermediate elements;
      end case
  end for
  $I_t = I$; "I becomes the target instance"
end
```

Fig. 3. An example of basic procedure

## 4 A practical example

In this section we consider an XML data set $D$ and its scheme $S$ expressed in XML Schema and perform on them two translations. First $S$ and $D$ are translated into a DTD $S'$ and an instance $D'$ valid for $S'$. Then, $S'$ and $D'$ are translated into the relational model. The input data and scheme are taken from an XML Query Use Case [12] and are reported in Figure 4.

The source scheme $S$ is first translated into the supermodel and the output is reported on the left hand side of Figure 5. This task is rather easy: each construct is just represented in terms of the corresponding primitive of the supermodel, each of which corresponds to a metaprimitive of the metamodel. For instance, a distinction has been made between complex elements, which have a structure, and atomic elements, which just have a base type associated with them.

The following step corresponds to the execution of the Algorithm reported in Figure 2. The output is reported on the right hand side of Figure 5. Three main transformations are performed as highlighted in the figure.

- An unordered sequence has been converted into an ordered one, since it is not possible to express this primitive in a DTD. This transformation yields also a modification on the source instance to preserve the order between elements in the sequence, as shown on the right hand side of Figure 6.
- Integers and decimals have been transformed into strings, because this is the only basic type available in a DTD.
- The cardinality constraint (1,10) has been converted into (1,N), since it is not possible to express a specific maximum value in a DTD.
Fig. 4. An XML Schema and one of its instances

Fig. 5. The representation of the scheme in Figure 4 into the supermodel and its translation into a format suitable for a DTD
The second translation corresponds to the execution of Algorithm 1 on the DTD and the XML data set generated by the first translation. The output is reported on the left hand side of Figure 7. Several transformations have been performed. We comment some of them.

- Complex elements have been turned into relations and atomic elements have been transformed into relational attributes.
- Keys have been introduced for each relation, as required by the relational model. This transformation involves a modification on the source instance: the procedure generates new identifiers, as shown in the output instance on the right hand side of Figure 7.
- Relations (like author and editor) have been unnested, according to the procedure described in Section 3.3.
- More appropriate base types have been associated with attributes. This information can be retrieved from the residual of the first translation, in which the base types of the original scheme have been stored.

We finally recall that, at the end of the algorithm, the output scheme and instance can be easily serialized into a format suitable for an external system. Alternatively, they can be expressed as a sequence of SQL statements for the definition of tables and for the population of the database.
Fig. 7. The target data of the second translation and the corresponding scheme in relational format

5 Related work

The problem of data translation is one of the issues that may arise when there is the need to combine heterogeneous sources of information in a coordinated and unified way. These include data integration [4, 22], schema matching [30], schema merging [28], and database federation [32]. Recently, Bernstein set the various problems within a very general framework that he called model management [5, 6]. In [7] the authors show, by means of practical examples, the value of model management as a methodology for approaching several metadata related problems with a significant reduction of programming effort.

According to [5], model management is based on five basic operators that, opportune ly combined, can be used to address the above problems:

- **Match**, which takes as input two schemes and returns a mapping between them,

- **Compose**, which takes as input two mappings between schemes and returns a mapping that combine them,

- **Merge**, which takes as input two schemes and returns a scheme that corresponds to their “union” and two mappings between the original schemes and the output scheme,

---

6 Note that we refer here to a “database” terminology that is different from the one used by Bernstein and by OMG [26]: they actually use the term model to denote our notion of scheme, metamodel to denote our notion of model, and metamodel to denote our notion of metamodel.
• **Diff**, which takes as input two schemes and a mapping between them and returns the subset of the first scheme not involved in the mapping, and

• **ModelGen**, which takes as input a scheme in a source model and a target model and returns the translation of the scheme into the target model.

Our proposal fits in this framework since it actually refers to an extension of ModelGen in that we also translate schema instances from one model to another.

Several studies have been conducted on model management, even if the majority of them concentrates only on specific operators. The Compose operator has been investigated in [23]. Rondo [24] is a rather complete framework for Model Management, but it does not address the problem of model translation. Cupid [23] focus on the Match operator, whereas Clio [25] provides an implementation of both the Match and of the Merge operators. The latter operator has also been studied in [28].

There were just a few attempts to implement the ModelGen operator. Bowers and Del-cambre [9] proposed Uni-Level Description, a framework for the management of multiple data models that makes use of a representation inspired by the metamodel proposed by Atzeni and Torlone in [3] and revised for Web data in [35]. This system is able to translate schemes and data between specific data models, making use of a set of transformation rules based on Datalog. Differently from our approach, these translations have to be defined for each pair of models (e.g., from the E-R model to the relational one).

Song et al. have proposed a theoretical approach to the implementation of model management operators based on graph grammars [33, 34]. Schemes and mappings are represented by graphs with the intention of simplifying user's interaction. The authors investigate the translation of schemes according to the ModelGen operator, but their approach requires as input a predefined mapping between the source schema and the target.

It should be said that many studies have been done on issues related to data translation, but they are often limited to a specific problem (e.g., the XML-relational mapping [18]) or to specific data models (survey papers on this subject can be found in [31]). Indeed, there have been some attempts to set the problem in a general framework [2, 8, 14, 29]. The main difference between our approach and these works relies on our notion of metamodel that introduces an higher level of abstraction with two main benefits. On the one hand, it provides a very natural way to describe heterogeneous models and, on the other hand, it allows us to define "generic" transformations between primitives that are independent of the specific models involved in a translation.

6 Conclusion and future work

In this paper, we have presented an approach to the translation of Web data between heterogeneous formats. Translations operate over XML representations of schemes and instances and are derived automatically by combining a number of predefined basic procedures performing XML transformations. The overall approach relies on a uniform description of models that we call metamodel.

From a practical point of view, we have developed a prototype of a rather complex tool for data translation (a preliminary version of this system has been presented in [36]). The tool manipulates XML representations of models with DOM (a platform-independent interface
that allows programs to dynamically access and update XML data) and performs translations by means of iterative XML transformations expressed in XQuery [17] and XSLT [13] over materialized temporary results. Currently, the system we have developed is able to fully translate schemes and data between several models (XML Schema and some of its dialects [21], DTD, Entity-Relationship model, Relational model, and ODL among others) and we are extending the tool with other formalisms and models for Web data (e.g., WebML [11]).

From a conceptual point of view we are currently investigating a number of properties related to the evaluation of the quality of a translation and we are studying techniques for generating “optimal” translations with respect to these properties.

References

18. D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. IEEE Data